# NODE.JS NIGHTS

STRV

# ARCHITECTURE

Jirka Erhart, Backend developer at STRV

STRV

## AGENDA

- Describe basic project architecture
- Implement authentication

## PROJECT ARCHITECTURE

“All problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection.“

David Wheeler

## REQUEST LIFECYCLE

Each request goes through certain steps while being served.

- Data parsing
- Data validation
- Permission validation
- Business logic execution
- Response data mapping

Our job is to clearly split responsibility for these steps among our code.

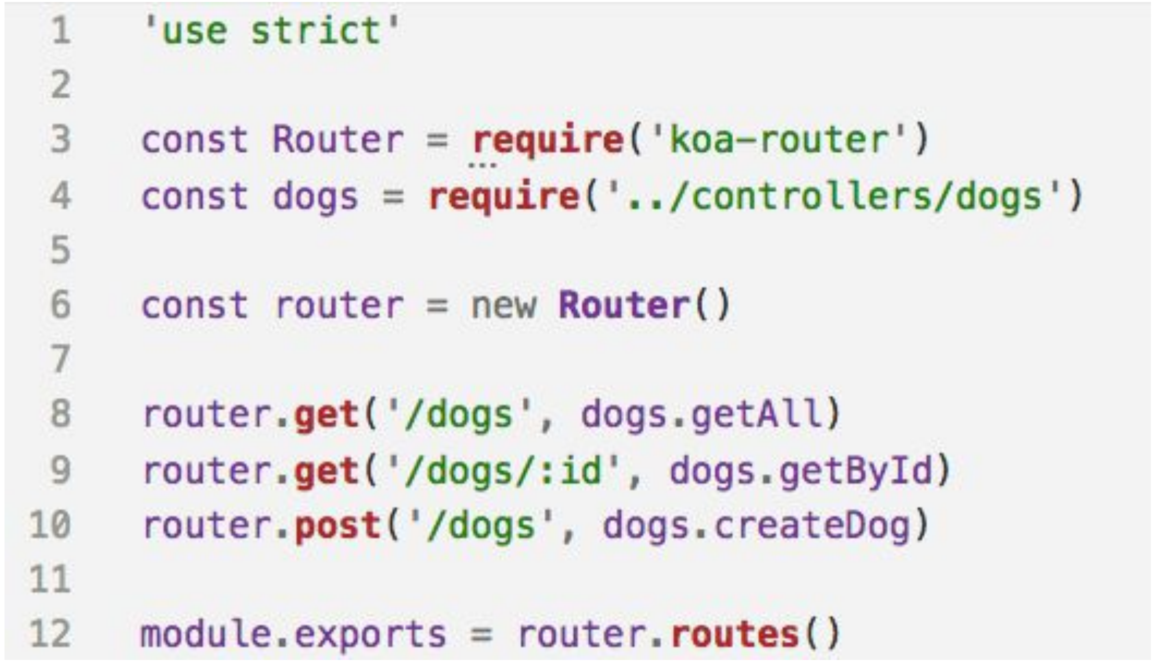
## DIVISION OF RESPONSIBILITIES

1) **app.js** (server composition)
2) **ROUTES** (routes definition)
3) **CONTROLLERS** (data validation, response mapping)
4) **OPERATIONS** (business logic execution)
5) **REPOSITORIES** (permanent storage access layer)

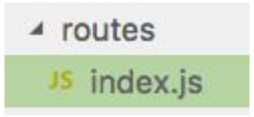
The rule is you only call directly underlying layer.

# ROUTES

- Define routes and bind them with controllers



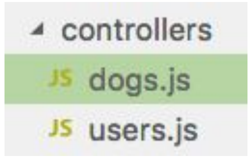
```
'use strict'

const Router = require('koa-router')
const dogs = require('../controllers/dogs')

const router = new Router()

router.get('/dogs', dogs.getAll)
router.get('/dogs/:id', dogs.getById)
router.post('/dogs', dogs.createDog)

module.exports = router.routes()
```

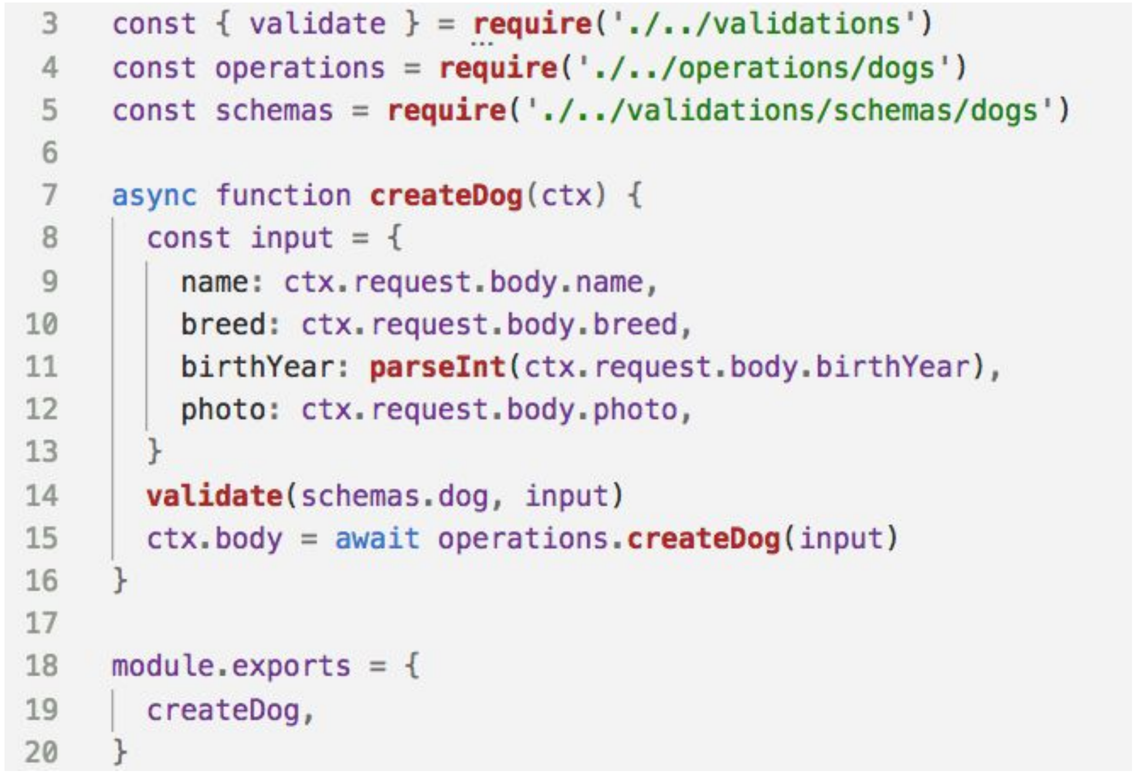


```
routes
  index.js
```

## CONTROLLERS

- Parse request data
- Validate request data
- Call operation(s)
- Set response



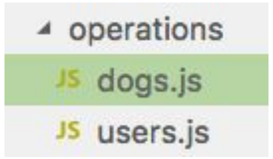


```
const { validate } = require('./../validations')
const operations = require('./../operations/dogs')
const schemas = require('./../validations/schemas/dogs')

async function createDog(ctx) {
  const input = {
    name: ctx.request.body.name,
    breed: ctx.request.body.breed,
    birthYear: parseInt(ctx.request.body.birthYear),
    photo: ctx.request.body.photo,
  }
  validate(schemas.dog, input)
  ctx.body = await operations.createDog(input)
}

module.exports = {
  createDog,
}
```

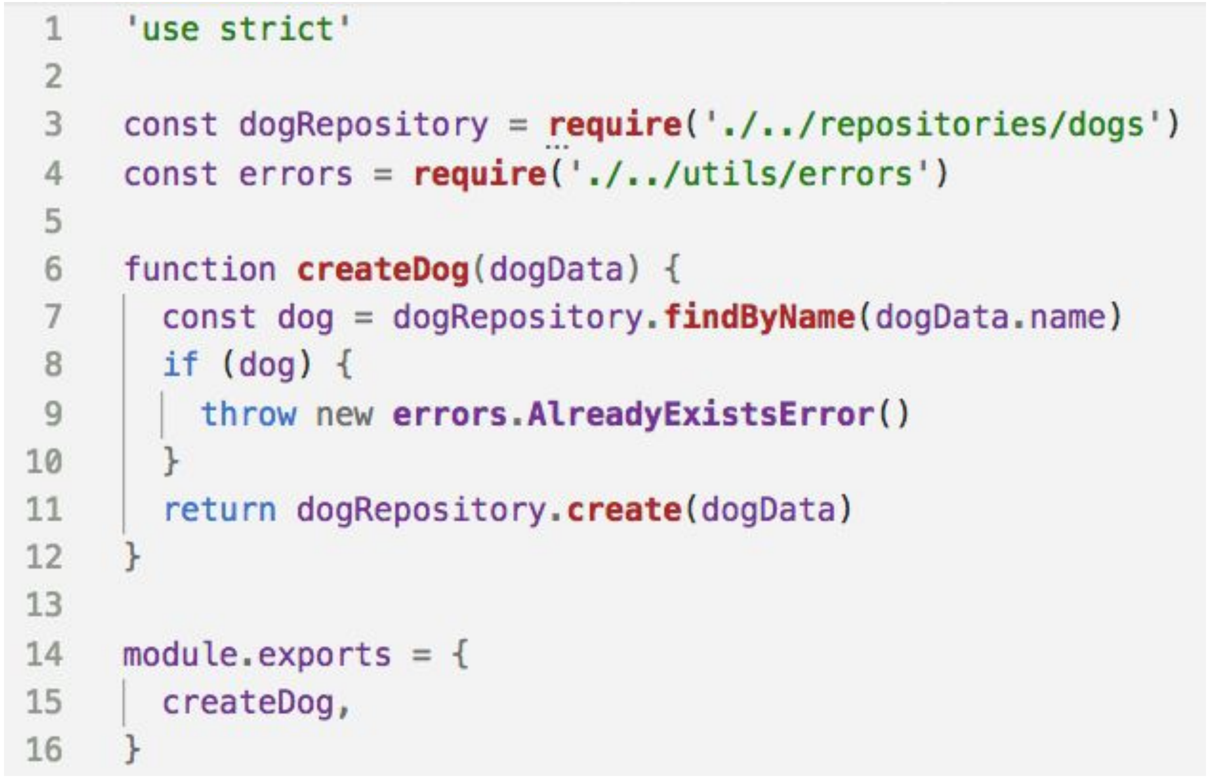
STRV

## OPERATIONS

- Perform business logic
- Throw errors



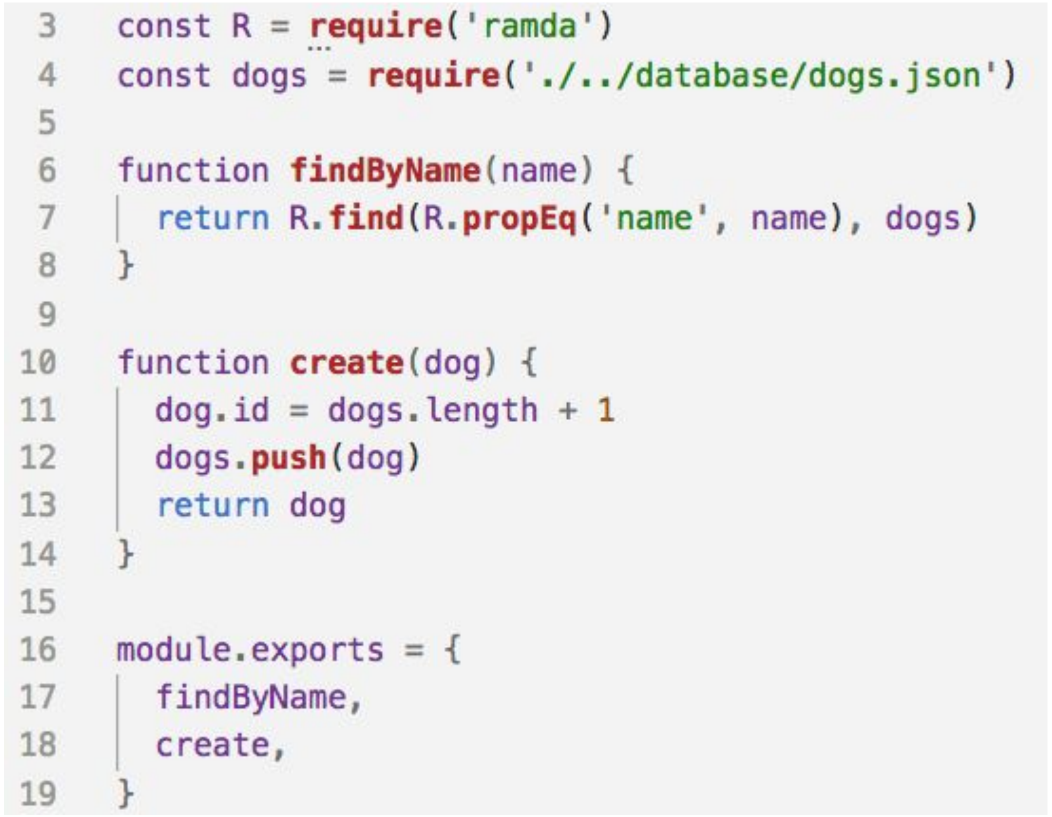
```
▴ operations
  JS dogs.js
  JS users.js
```



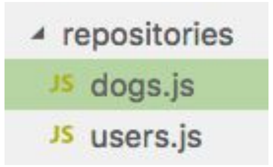
```
'use strict'

const dogRepository = require('./../repositories/dogs')
const errors = require('./../utils/errors')

function createDog(dogData) {
  const dog = dogRepository.findByName(dogData.name)
  if (dog) {
    throw new errors.AlreadyExistsError()
  }
  return dogRepository.create(dogData)
}

module.exports = {
  createDog,
}
```

# REPOSITORIES

- Abstract from specific database/ORM
- Simplify database calls for operations



```javascript
const R = require('ramda')
const dogs = require('./../database/dogs.json')

function findByName(name) {
  return R.find(R.propEq('name', name), dogs)
}

function create(dog) {
  dog.id = dogs.length + 1
  dogs.push(dog)
  return dog
}

module.exports = {
  findByName,
  create,
}
```



```
◢ repositories
  JS dogs.js
  JS users.js
```

STRV

WE NEED TO GO
DEEPER

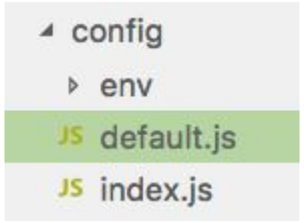
## DIVISION OF RESPONSIBILITIES

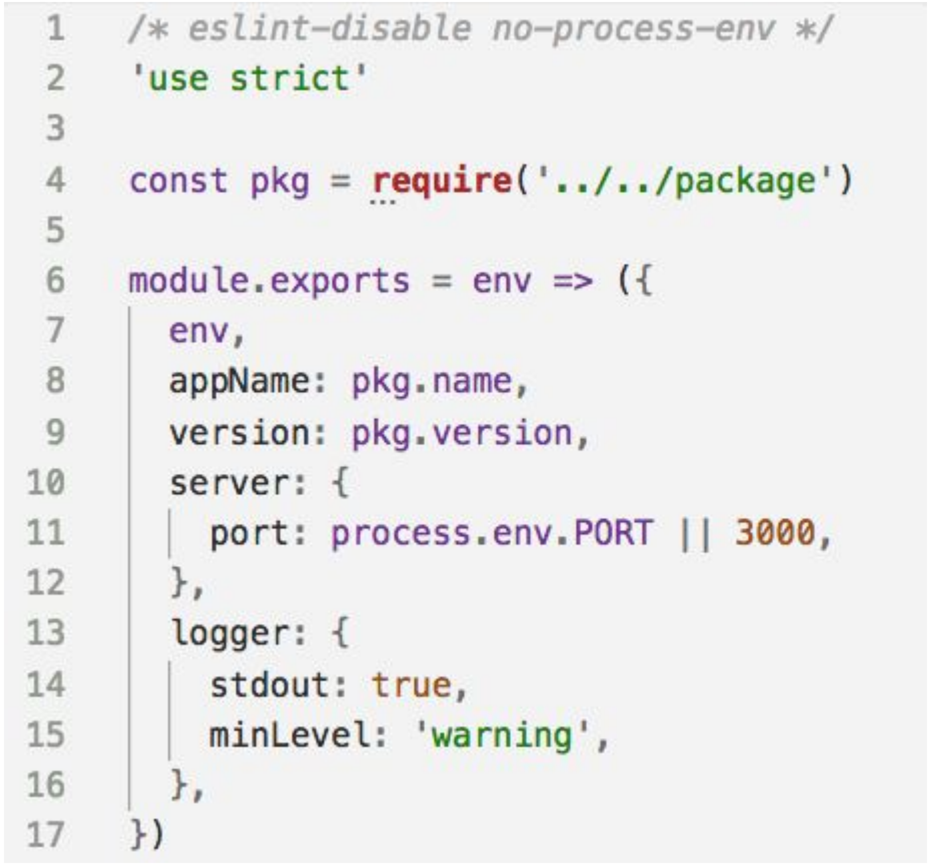
- **CONFIG** (application-wide configuration)
- **DATABASE** (database models)
- **MIDDLEWARE** (logic concerning all requests)
- **UTILS** (you know, that other stuff..)
- **VALIDATIONS** (validation schemas and composition)

## CONFIGURATION

## DEFAULT CONFIG

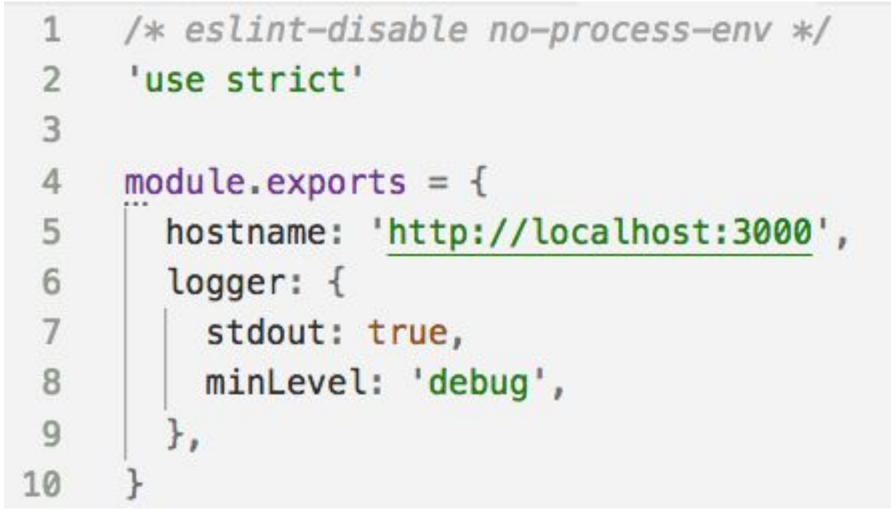
- Sets default configuration


config
env
default.js
index.js

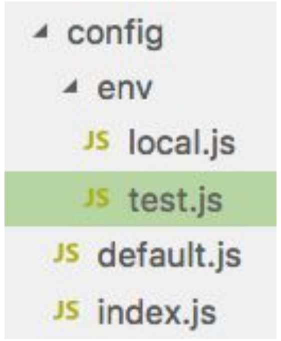

```
/* eslint-disable no-process-env */
'use strict'

const pkg = require('../../package')

module.exports = env => ({
  env,
  appName: pkg.name,
  version: pkg.version,
  server: {
    port: process.env.PORT || 3000,
  },
  logger: {
    stdout: true,
    minLevel: 'warning',
  },
})
```

# ENVIRONMENT CONFIG

- Overrides default configuration for specific environment



```
/* eslint-disable no-process-env */
'use strict'

module.exports = {
  hostname: 'http://localhost:3000',
  logger: {
    stdout: true,
    minLevel: 'debug',
  },
}
```

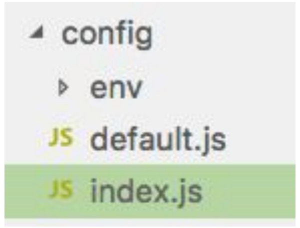


```
◢ config
  ◢ env
      local.js
      test.js
    default.js
    index.js
```

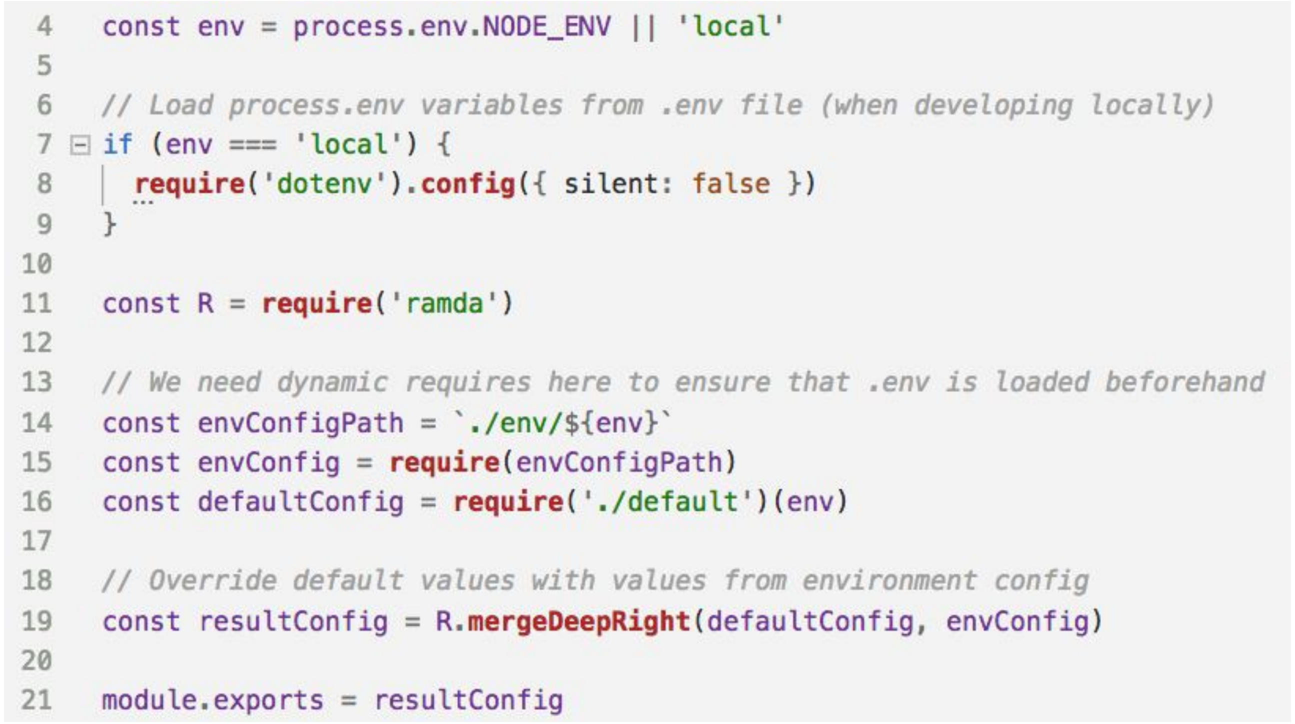
# CONFIG COMPOSITION

Composes configuration from

1) Env. variables
2) Env. config
3) Default config



```
▴ config
  ▸ env
  default.js
  index.js
```



```javascript
const env = process.env.NODE_ENV || 'local'

// Load process.env variables from .env file (when developing locally)
if (env === 'local') {
  require('dotenv').config({ silent: false })
}

const R = require('ramda')

// We need dynamic requires here to ensure that .env is loaded beforehand
const envConfigPath = `./env/${env}`
const envConfig = require(envConfigPath)
const defaultConfig = require('./default')(env)

// Override default values with values from environment config
const resultConfig = R.mergeDeepRight(defaultConfig, envConfig)

module.exports = resultConfig
```

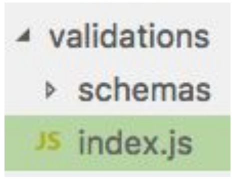
STRV

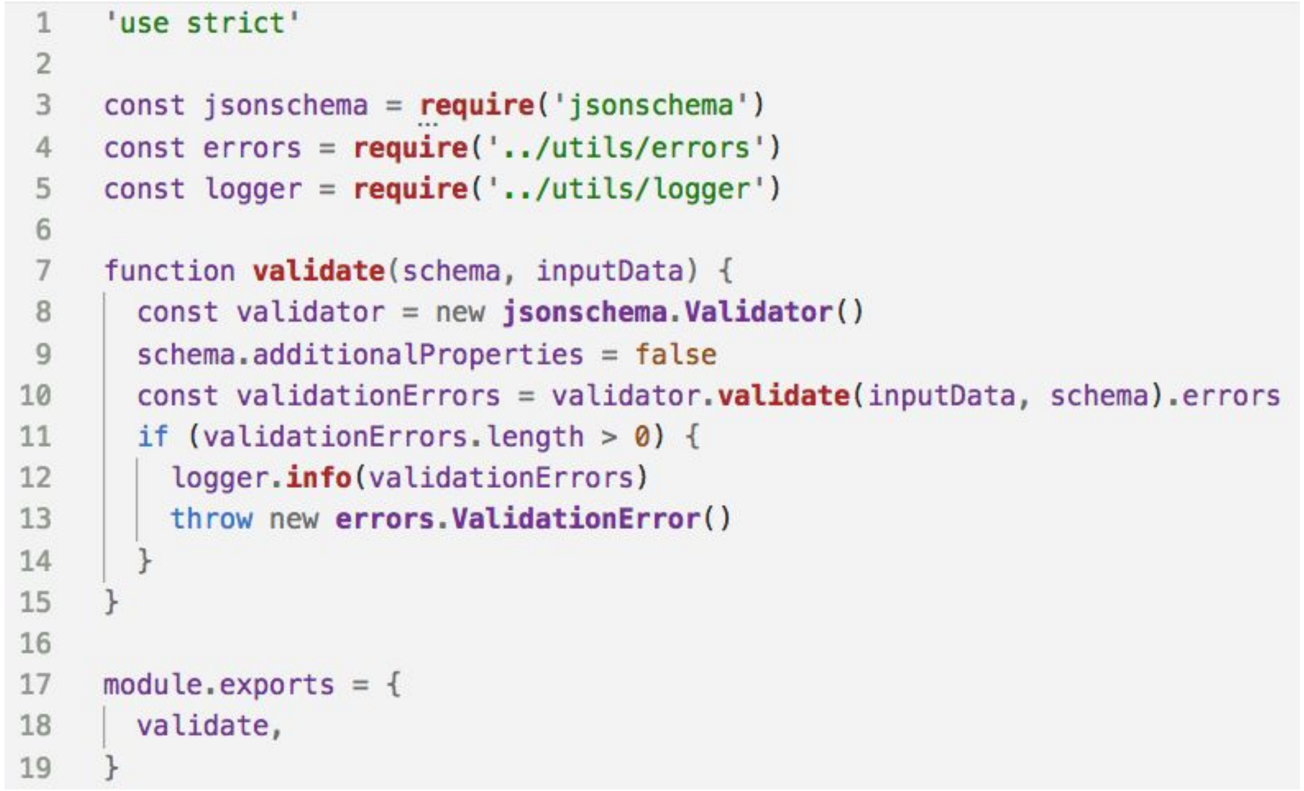
# VALIDATION

# VALIDATIONS

Validate schema against input data

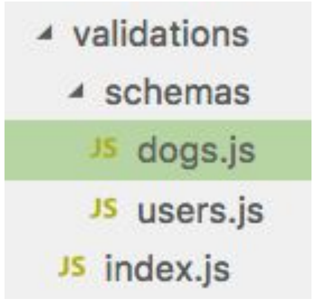
- Be strict
- Throw error on fail


validations
schemas
index.js

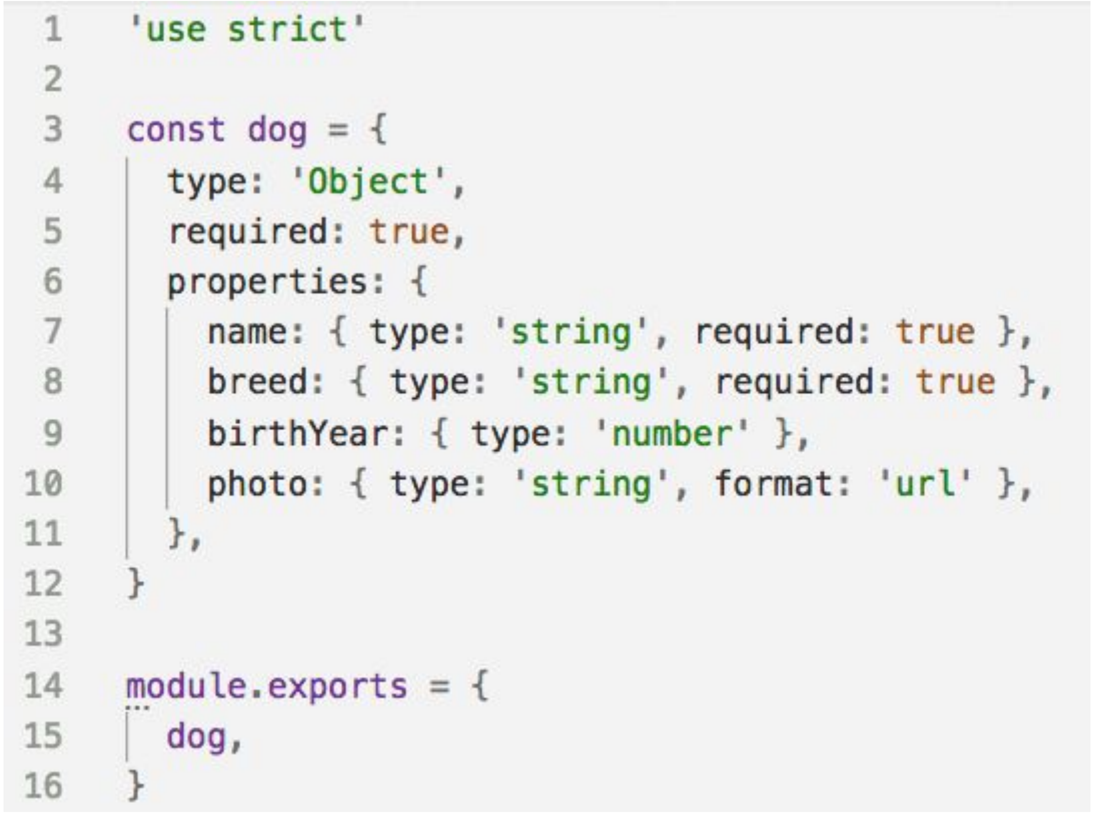

```javascript
'use strict'

const jsonschema = require('jsonschema')
const errors = require('../utils/errors')
const logger = require('../utils/logger')

function validate(schema, inputData) {
  const validator = new jsonschema.Validator()
  schema.additionalProperties = false
  const validationErrors = validator.validate(inputData, schema).errors
  if (validationErrors.length > 0) {
    logger.info(validationErrors)
    throw new errors.ValidationError()
  }
}

module.exports = {
  validate,
}
```

## VALIDATION SCHEMAS

- Schemas grouped by entity
- Should be as strict as possible



```
validations
  schemas
    dogs.js
    users.js
  index.js
```

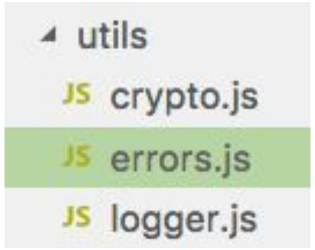


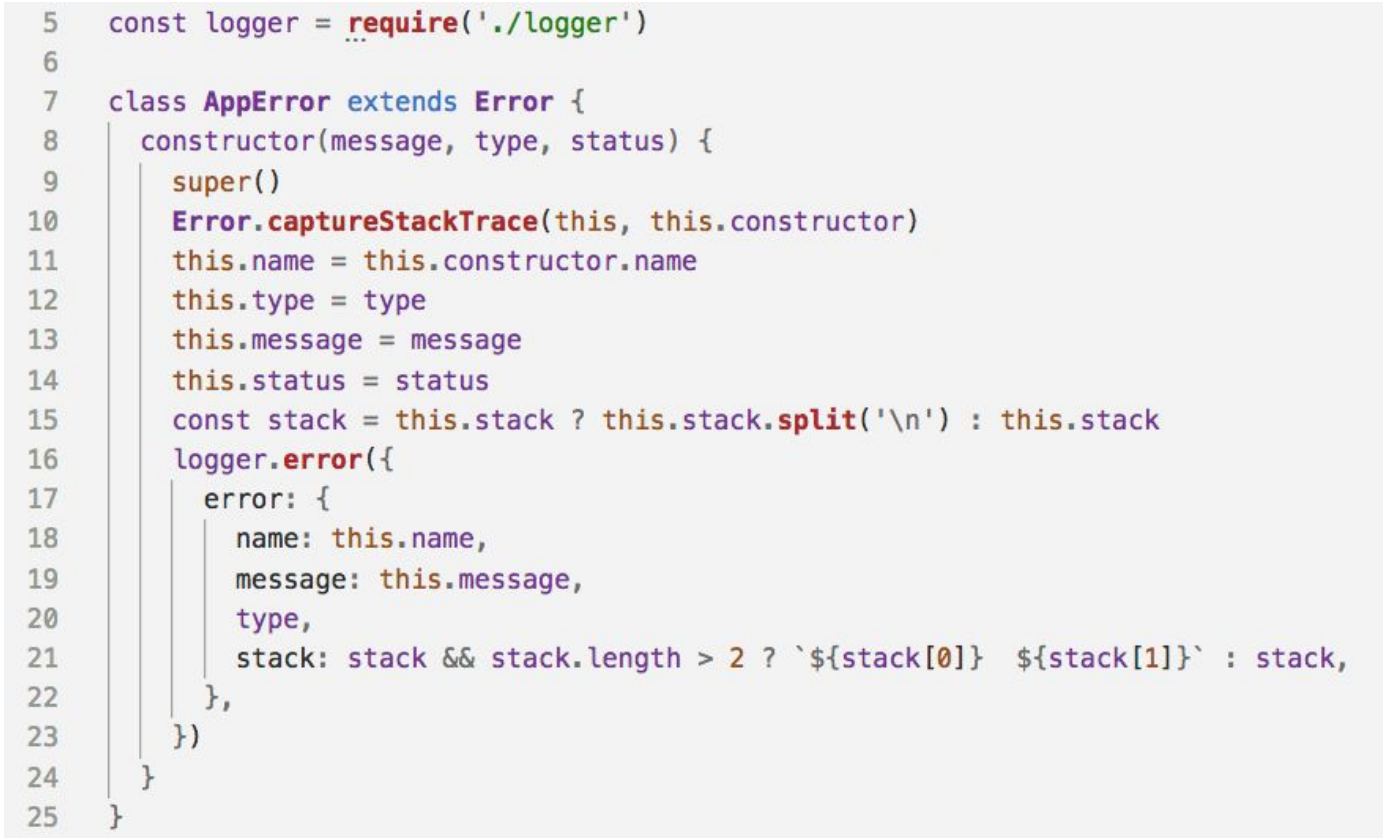
```
'use strict'

const dog = {
  type: 'Object',
  required: true,
  properties: {
    name: { type: 'string', required: true },
    breed: { type: 'string', required: true },
    birthYear: { type: 'number' },
    photo: { type: 'string', format: 'url' },
  },
}

module.exports = {
  dog,
}
```

## ERRORS

# ERRORS

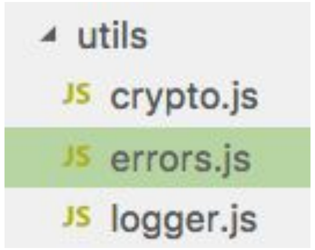
Have common ancestor to ease use


utils
crypto.js
errors.js
logger.js

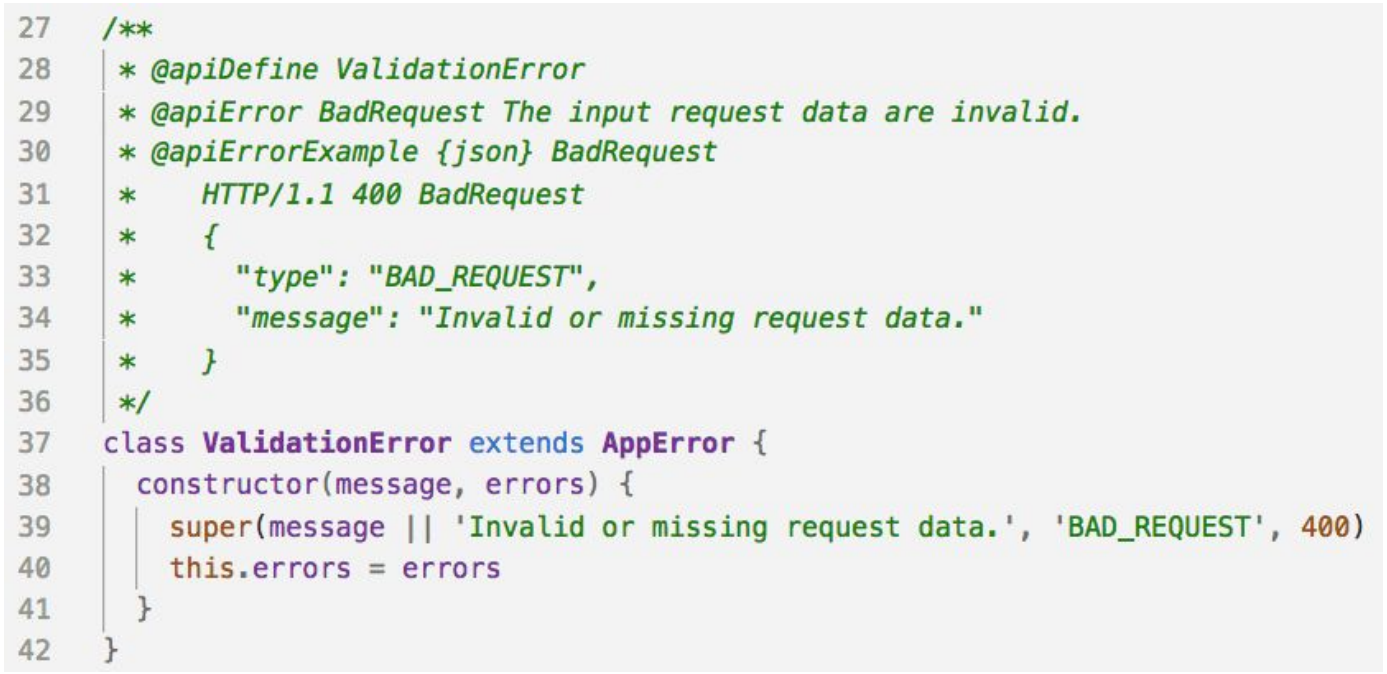

```
const logger = require('./logger')

class AppError extends Error {
  constructor(message, type, status) {
    super()
    Error.captureStackTrace(this, this.constructor)
    this.name = this.constructor.name
    this.type = type
    this.message = message
    this.status = status
    const stack = this.stack ? this.stack.split('\n') : this.stack
    logger.error({
      error: {
        name: this.name,
        message: this.message,
        type,
        stack: stack && stack.length > 2 ? `${stack[0]}  ${stack[1]}` : stack,
      },
    })
  }
}
```

# ERRORS

And many specific descendants



```
utils
  crypto.js
  errors.js
  logger.js
```



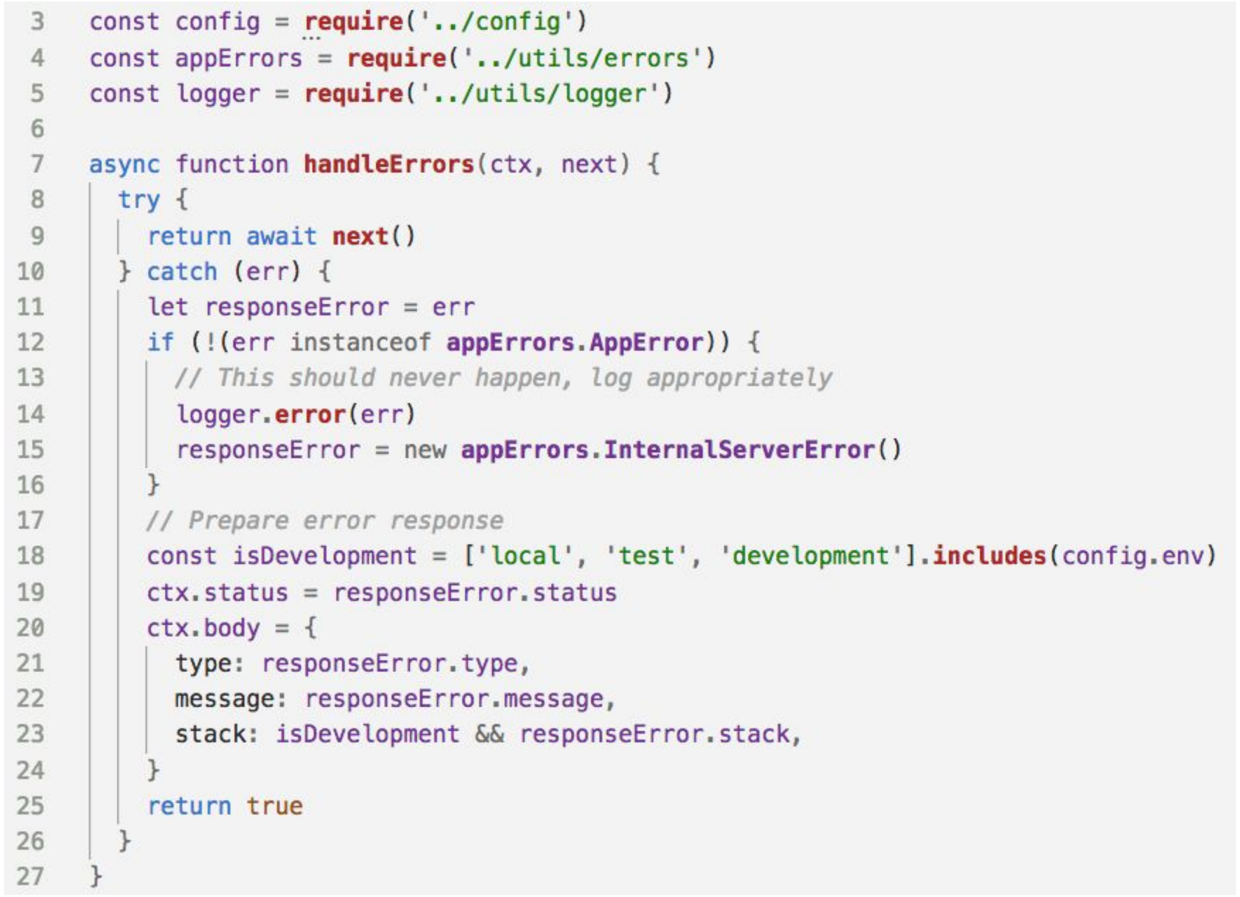
```
/**
 * @apiDefine ValidationError
 * @apiError BadRequest The input request data are invalid.
 * @apiErrorExample {json} BadRequest
 *     HTTP/1.1 400 BadRequest
 *     {
 *       "type": "BAD_REQUEST",
 *       "message": "Invalid or missing request data."
 *     }
 */
class ValidationError extends AppError {
  constructor(message, errors) {
    super(message || 'Invalid or missing request data.', 'BAD_REQUEST', 400)
    this.errors = errors
  }
}
```

STRV

# ERROR HANDLING

Handle in middleware as it concerns all requests

Convert errors to error responses

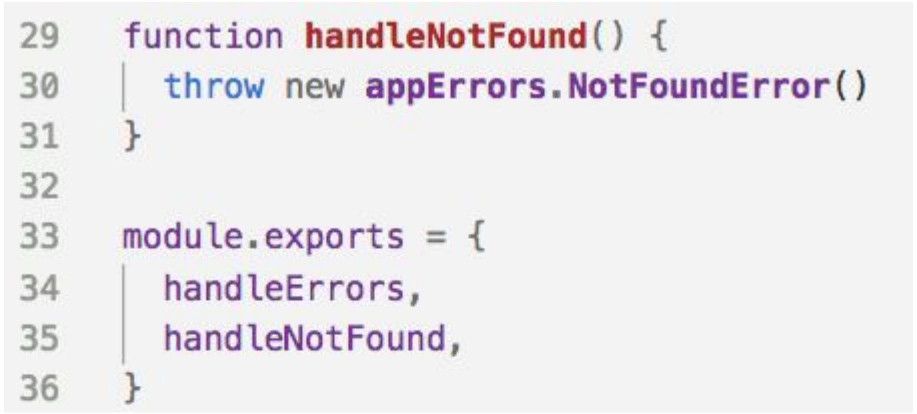


```
const config = require('../config')
const appErrors = require('../utils/errors')
const logger = require('../utils/logger')

async function handleErrors(ctx, next) {
  try {
    return await next()
  } catch (err) {
    let responseError = err
    if (!(err instanceof appErrors.AppError)) {
      // This should never happen, log appropriately
      logger.error(err)
      responseError = new appErrors.InternalServerError()
    }
    // Prepare error response
    const isDevelopment = ['local', 'test', 'development'].includes(config.env)
    ctx.status = responseError.status
    ctx.body = {
      type: responseError.type,
      message: responseError.message,
      stack: isDevelopment && responseError.stack,
    }
    return true
  }
}
```

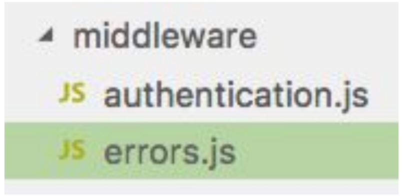
# ERROR HANDLING

We also need to handle not found error

We do so in the same file



```
function handleNotFound() {
  throw new appErrors.NotFoundError()
}

module.exports = {
  handleErrors,
  handleNotFound,
}
```

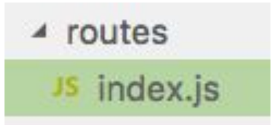


```
middleware
  authentication.js
  errors.js
```

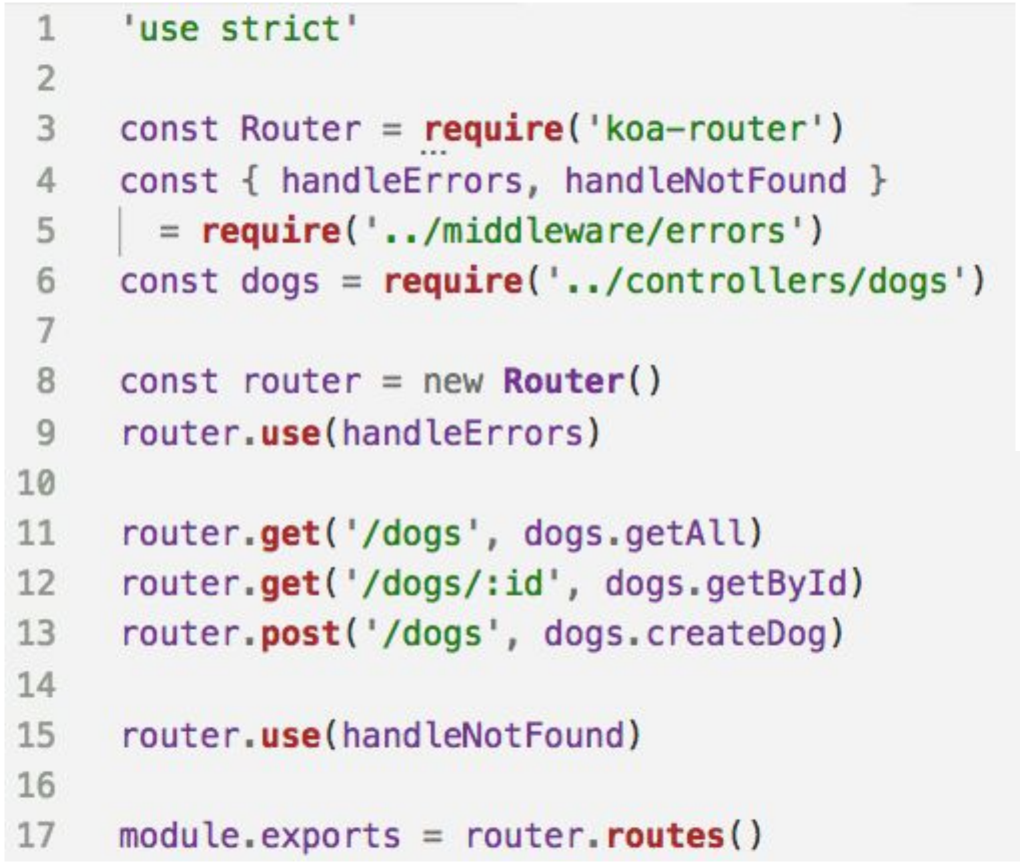
STRV

## ERROR HANDLING

And finally we use the middleware in routes



```
▲ routes
   index.js
```



```javascript
'use strict'

const Router = require('koa-router')
const { handleErrors, handleNotFound }
  = require('../middleware/errors')
const dogs = require('../controllers/dogs')

const router = new Router()
router.use(handleErrors)

router.get('/dogs', dogs.getAll)
router.get('/dogs/:id', dogs.getById)
router.post('/dogs', dogs.createDog)

router.use(handleNotFound)

module.exports = router.routes()
```

# QUESTIONS?

## REFRESHMENTS

🎉

STRV

## LET’S GET TO BUSINESS

STRV

## WHAT WE'VE LEARNED TODAY

- API code organisation
- Environment aware configuration
- Application wide error handling
- Stateless user authentication

## HOMEWORK

- Update your dog CRUD operations to match presented architecture
- Implement user sign in (using presented architecture)

# THAT'S IT

Jirka Erhart
jiri.erhart@strv.com

STRV

## QUESTIONS

STRV

STRV